More Reasoning

Michael P. Fourman

February 2, 2010

1 Structural Induction

In previous lecture notes we have seen how to carry out inductive proofs on lists and trees. To re-cap, here are the datatype definitions, and the associated inductive proof rules:

```
datatype 'a list = nil
                 | :: of 'a * 'a list
```

$$\left(\begin{array}{l}\phi(\texttt{nil})\ \wedge \\ \forall h, t \cdot \phi(t) \rightarrow \phi(h::t)\end{array}\right) \rightarrow \forall l \cdot \phi(l)$$

```
datatype 'a tree = Leaf
                 | Node of 'a tree * 'a * 'a tree;
```

$$\left(\begin{array}{l}\phi(\texttt{Leaf})\ \wedge \\ \forall v, t_1, t_2 \cdot (\phi(t_1) \wedge \phi(t_2)) \rightarrow \phi(\texttt{Node}(t_1, v, t_2))\end{array}\right) \rightarrow \forall t \cdot \phi(t)$$

You should be able to start noticing a pattern here. In both datatypes we can build larger instances of the structure by applying a constructor to smaller instances (using `::` and `Node`). There are also constructors that act as a base case (`nil` and `Leaf`). The format of the associated induction rules mirrors the way these structures are built. Let's examine the datatype `bexp`, which represents simple Boolean expressions.

```
datatype Bexp = V of string
              | /\ of Bexp * Bexp
              | \/ of Bexp * Bexp
              | ~  of Bexp;
```

This datatype has more constructors than trees or lists, but the basic idea is the same. We have some constructors for building expressions out of existing

types (in this case `V`) and some constructors for building expressions out of smaller expressions (in this case `/\`, `\/` and `~`). The format of the associated inductive rule shouldn't be too surprising:

$$\left(\begin{array}{l} \forall s \cdot \phi(\texttt{V s}) \ \wedge \\ \forall c_1, c_2 \cdot (\phi(c_1) \wedge \phi(c_2)) \rightarrow \phi(c_1 \ \texttt{/\ } \ c_2) \ \wedge \\ \forall c_1, c_2 \cdot (\phi(c_1) \wedge \phi(c_2)) \rightarrow \phi(c_1 \ \texttt{\textbackslash/} \ c_2) \ \wedge \\ \forall c \cdot \phi(c) \rightarrow \phi(\texttt{\~{}}c) \end{array}\right) \rightarrow \forall c \cdot \phi(c)$$

You should be able to follow the pattern to produce induction rules for many datatypes of this form. Try to do this for some of the variations of the tree datatype illustrated in earlier notes. This kind of induction is known as *structural induction*. Most finite datatypes admit this kind of inductive proof technique, although in some cases the form of the inductive rule can be a bit messy. For example, consider a tree datatype where we allow an arbitrary number of children at each node:

```
datatype 'a tree = Node of 'a * ('a tree list);
```

Can you write down the structural induction rule for this type?

The technique does not work on datatypes built from functional values. For example, consider

```
datatype funny = F of funny -> funny;
```

We cannot get a structural induction rule for this datatype, and so we cannot use induction to prove that a property ϕ holds for all objects of type `funny`. Dealing with datatypes such as this is outside the scope of the course. However, most datatypes do not make use of such functional values, and so this limitation is not particularly serious.

2 Well-founded induction

A binary relation $\prec$ is *well-founded* if there exist no infinite decreasing chains

$$\ldots \prec x_n \prec \ldots \prec x_2 \prec x_1$$

The ordering $<$ on the natural numbers is well-founded, but $<$ on the integers isn't, and neither is $<$ on the rational numbers. The following infinite chains provide the evidence:

$$\ldots < -n < \ldots < -2 < -1 \qquad \text{and} \qquad \ldots < \frac{1}{n} < \ldots < \frac{1}{2} < \frac{1}{1}$$

Note that we can't just say that $<$ is well-founded — we have to state the domain of the relation. Another example of a well-founded relation is the

lexicographic ordering of pairs of natural numbers, defined by

$$(i', j') \prec_{\text{lex}} (i, j) \text{ if and only if } i' < i \ \vee \ ((i' = i) \wedge j' < j)$$

We can extend this ordering to tuples, quadruples and so forth. If f is a function into the natural numbers, then there is a well-founded relation $\prec_f$ defined by

$$x \prec_f y \text{ if and only if } f(x) < f(y)$$

A function f used for this purpose is known as a *measure function.*

Why all this interest in well-founded relations? Suppose $\prec$ is a well-founded relation over some type α, and $\phi(x)$ a property to be proved for all x of type α. The technique of *well-founded* induction can be used to prove this. The induction rule may be written as

$$\left(\begin{array}{l} \forall y. \\ \qquad (\forall x \cdot \ x \prec y \rightarrow \phi(x)) \rightarrow \phi(y) \end{array} \right) \rightarrow \forall x \cdot \ \phi(x)$$

In other words it suffices to prove, for all y, the following induction step:

$$\text{if } \phi(x) \text{ for all } x \prec y \text{ then } \phi(y)$$

All of the induction rules we have seen up to this point can be obtained from this rule by suitable choice of the well-founded relation. If $\prec$ is

- $<$ on the natural numbers we get complete induction.
- $\prec_N$, where $m \prec_N n$ just if $m + 1 = n$ then we get (after some simplification) mathematical induction.
- $\prec_L$, where $l \prec_N l'$ just if $h :: l = l'$ for some h, then we get structural induction on lists.

A well-founded relation given by a measure function yields induction on the size of an object.

3 Well-founded recursion

Let $\prec$ be a well-founded relation over some type α. If f is a function with formal parameter x that makes recursive calls $f(y)$ only if $y \prec x$, then $f(x)$ terminates for all x. In this case, we say that f is defined by *well-founded recursion* on $\prec$. Informally, $f(x)$ terminates because, since there are no infinite decreasing chains in $\prec$, there can be no infinite recursion.

Proving that a function terminates suggests a useful form of induction for it. If a function is defined by well-founded recursion on $\prec$, then its properties can often be proved by well-founded induction on $\prec$.

4 Reasoning about Functionals

We have seen how functionals can be used to write very concise programs. We would like to be able to prove some properties of such functionals. For example, if we have two functions, `f` and `g`, and we map `g` over a list `l` and then map `f` over the result, then we would expect to get the same result if we map `f o g` over `l`, i.e. we would like to prove that

```
(map f) o (map g) = map (f o g)
```

Before we attempt to prove such an equivalence we need to clarify what we mean by equality here. The *law of extensionality* states that functions f and g are equal if $f(x) = g(x)$ for all x (of suitable type). The extensionality law is valid in ML because the only operation that can be performed on an ML function is application to an argument. If two functions are extensionally equal, then replacing one by the other doesn't affect the final result (we are assuming for simplicity that all functions terminate). Some languages (notably LISP) regard two functions as being equal only if their definitions are identical. LISP uses this *intensional equality* because it views a function value as a piece of code that can be taken apart. If we stick to extensional equality we can prove that `(map f) o (map g)` is indeed equal to `map (f o g)`. We just have to apply both sides of the equality to an arbitrary list l and then simplify. Now `(map f) o (map g) l = map f (map g l)` and so we must prove that for all `l`,

```
map f (map g l) = map (f o g) l
```

Structural induction on `l` can be used to prove this.

Such equivalences are important to a compiler writer. We might express our algorithm in terms of two separate maps for clarity. This is often the case when the functions `(map f)` and `(map g)` perform interesting tasks in their own right. However, it is clearly inefficient to first map `g` over a list and then to map `f` over the result. It is better to map `f o g` over the list in one go. Proving such equivalences allows a compiler writer to transform programs to more efficient ones whilst guaranteeing that the observable behaviour of the program remains unchanged.